[illegible]

NPSCS-91-003

NAVAL POSTGRADUATE SCHOOL
Monterey, California

ANALYTICAL DERIVATION OF SOFTWARE FAILURE REGIONS

Timothy J. Shimeall
MAJ John Manning Bolchoz, USA
CDR Rachel Griffin, USN

September 1991

Prepared for:

Naval Weapons Center
China Lake, CA 93555-6001

This report was prepared for the Naval Weapons Center and funded by the Naval Postgraduate School.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPSCS-91-003	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL *(if applicable)* CS	7a. NAME OF MONITORING ORGANIZATION Naval Weapons Center
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943	7b. ADDRESS *(City, State, and ZIP Code)* China Lake, CA 93555-6001
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School; 8b. OFFICE SYMBOL *(if applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER OM&N Direct Funding
8c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
ANALYTICAL DERIVATION OF SOFTWARE FAILURE REGIONS(U)

12. PERSONAL AUTHOR(S)
Shimeall, Timothy J., Bolchoz, John Manning, Griffin, Rachel

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Progress	FROM 10/90 TO 9/91	1991, Sept. 9	34

16. SUPPLEMENTARY NOTATION
The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the US Government.

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Software Testing,Formal Models, Software Faults, Software Failures, Software Tools

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*
This paper proposes an analytical method for deriving software failure regions, which are regions of the input space that are mapped to failures by specific faults. Previous studies have used empirical rather than analytical approaches to derive failure regions. A manual technique is presented and proven to produce the necessary and sufficient condtions of a fault being executed and leading to a failure. Semiautomated tools to assist in the manual technique are discussed, as is the use of failure regions in regression testing.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Timothy J. Shimeall	22b. TELEPHONE *(Include Area Code)* (408) 646-2509; 22c. OFFICE SYMBOL CS/Sm

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Analytical Derivation of Software Failure Regions[1]

Timothy J. Shimeall
MAJ John Manning Bolchoz, US Army
CDR Rachel Griffin, US Navy
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

Abstract

This paper proposes an analytical method for deriving software failure regions, which are regions of the input space that are mapped failures by specific faults. Previous studies have used empirical rather than analytical approaches to derive failure regions. A manual technique is presented and proven to produce the necessary and sufficient conditions of a fault being executed and leading to a failure. Semiautomated tools to assist in the manual technique are discussed, as is the use of failure regions in regression testing.
Index terms: Software Testing, Formal Models, Software Faults, Software Failures, Software Tools

1 Introduction

It is widely recognized that many faults in installed software are introduced during software maintenance. For example, a recent change to the switching software used in the US telephone system inadvertently introduced a three-line fault that interrupted service to customers in six states.[17] Regression testing is the process of testing software to determine if a modification has introduced a fault. Currently, there is no consistent theoretical basis for the conduct of regression testing, forcing maintainers to rerun all previous tests (in the phone system example, a 13-week process) to determine if any previously-correct results have been affected during maintenance. One reason for modifying software during maintenance is to correct known faults in the software. This paper describes and verifies a technique for predicting the domain of input that produces results affected by a specific fault in a specific piece of software. This technique is supported by a set of software tools,

[1]The research reported here was conducted with funds administered by the Naval Postgraduate School Research Council.

also described in this paper. Knowledge of this input domain focuses the testing needed to determine if correction of the fault corresponding to the domain has introduced further faults. For convenience of presentation, this paper assumes that the software to be analyzed is written in an ALGOL-derived language.

In this paper, a fault is an erroneous piece of program source code. An error is a discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition, the immediate result of the execution of a fault. A failure is the termination of the ability of a functional unit to perform its required function (i.e., a failure is either the production of a value that contradicts some portion of the specification, or the lack of production of a specified value).[1]

A software failure region is the set of all input values that are mapped by an individual program fault onto any failure. Each fault in a program, by definition, is associated with a single failure region, which may be composed of several disjoint or overlapping subregions. A single failure region may be associated with more than one fault, these faults being differentiated by varying effects on the program results. Different faults may also map distinct failure regions onto the same set of program failures. See figure 1 for a diagram of these failure region – fault – set of failure associations.

This paper describes a manual technique for analytically deriving a failure region from a known fault in a program's source code. This technique views a failure region as bounded by the conjunction of three boolean conditions. The first, *reachability*, states the conditions that hold when the program executes the section of source code containing the fault. The second, *generation*, states the conditions that cause the fault to produce an erroneous internal state (e.g., bad variable values). The third, *propagation*, states the conditions under which the erroneous internal state becomes a failure.

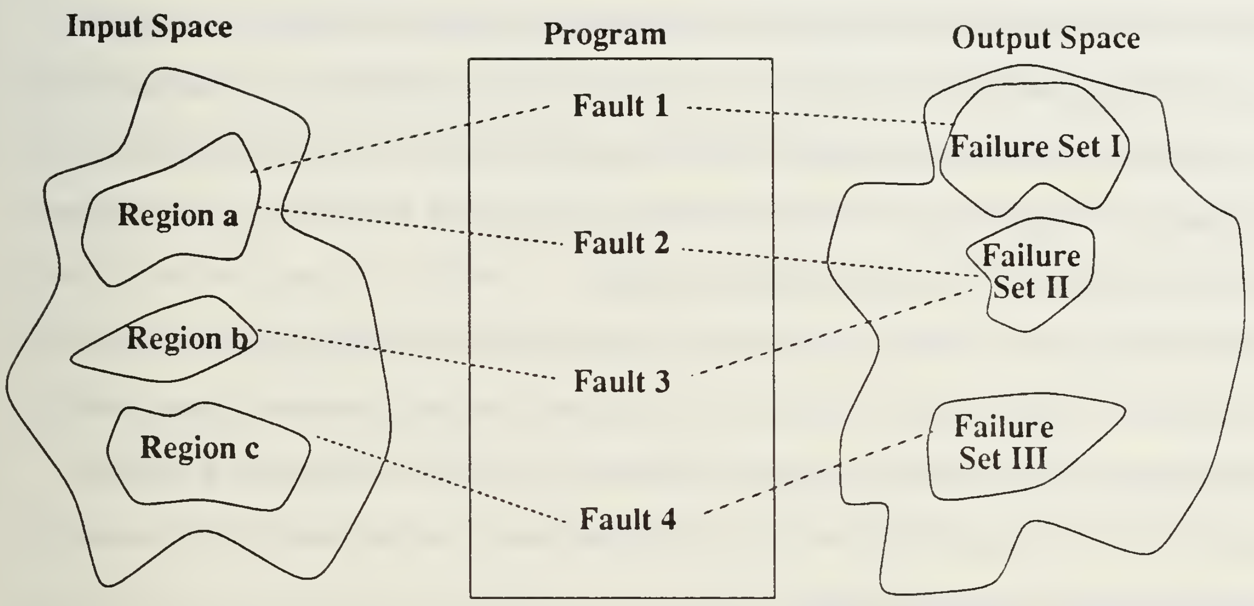


Figure 1: Associations between Failure Regions, Faults and Failures

If software developers can efficiently determine a software failure region, regression testing would be simplified. The bounding conditions of a failure region are the necessary and sufficient conditions for a failure due to the associated fault. The regression-testing subtask of evaluating if the fault is fully corrected may be viewed as applying a sample from the failure region to the program and examining the results for failures. Instead of simply repeating previous tests, testers could apply intelligent sampling techniques on the failure region. The regression-testing subtask of evaluating if the fault correction has not introduced new faults may also be facilitated by knowledge of the failure region. This latter question is the subject of ongoing research[6].

The following section briefly reviews research that has involved failure regions or concepts similar to failure regions. Section three describes a manual method for analytical derivation of the failure regions. Section four describes automated tools that have been developed to aid this analysis. This paper concludes with a discussion of specific applications of failure regions to regression testing.

2 Failure Regions

Software failure regions are sets of program inputs, one set for each fault in the program and each element in each set a complete series of inputs that causes the program to execute the corresponding fault and produce a failure due to the fault. These sets are always finite, given the limitations on representation in a finite machine, but are often too large for tractable enumeration. Each element of each set may be of complex structure. Due to these considerations, this paper will denote failure regions by the boolean conditions that identify their elements.

Ammann and Knight first described software failure regions in research on software fault tolerance[3]. They identified and characterized failure regions by repeated sampling across a plane through the program input space. They used plots of the identified portions of the failure regions as a basis for the proposal of a data-variation fault-tolerance technique. No attempt was reported to identify the bounding conditions of the failure regions. Dunham and Finelli applied this sampling procedure to other software[4], referring to failure regions to as 'error crystals'.

In parallel with this fault-tolerance-based interest in the effects of known faults, testing-based research has investigated conditions that would lead to detection of suspected or hypothesized faults[11, 18, 5, 12]. The testing techniques that have been derived from that research seek to identify at least one data set that would reveal a fault, rather than in deriving the necessary and sufficient conditions to identify all data that would reveal the fault. These two problems are clearly related, but they are not identical, and it is the latter problem that is the subject of this paper.

Voas[16] investigated the dual of the problem discussed in this paper, namely identification of the portions of the program *text* that are most likely to contain faults by examination of where faults are likely to be propagated or obscured. This probabilistic work has focussed principally on missing-statement faults and the insights from that work are currently under examination using software failure regions[6].

While there has been broad interest in the conditions that lead to fault detection, no one has previously proposed a technique to derive the necessary and sufficient conditions for a fault to cause a failure. Such a technique is the focus of the next section.

3 Manual Technique for Deriving Failure Regions

3.1 Assumptions

There are several assumptions that are implicit in the failure regions analysis technique. These assumptions may limit the applicability of the technique and are provided here to give context to the technique description that follows.

The first assumption is that the region is to correspond to a known fault. The region analysis technique does not detect faults, but generates failure regions for previously-detected faults. Myers[10] conjectured that the sorts of information that appear in a failure region's bounding conditions may be used to detect other faults, but this has yet to be empirically determined. The next section describes the characteristics of a fault used in application of the region analysis technique .

The second assumption is that, while deriving the failure region corresponding to a given fault, the effects of other faults may be ignored. In the early stages of program testing, this assumption may be quite restrictive, and research is anticipated to derive failure regions for multiple faults simultaneously.

The third assumption is that the analyst knows the loop effects for each loop in the program, whether iterative or recursive. Specifically, the technique assumes the analyst can indicate what changes in the local data state of a program execution are made by a loop. This may be easier to define then a loop invariant. While identification of loop effects is impossible in the general case, since not all loops terminate or can be proved to terminate, in many cases the source code, design or requirements documentation will readily provide this information.

The fourth assumption is that the program under consideration is a sequential program with soft-real-time constraints. The program is assumed sequential to avoid consideration of interleaving and interference effects. It is assumed under soft-real-time constraints to allow analysis for excessive looping faults, but to avoid consideration of explicit timing values during the analysis. Research is underway to extend the analysis technique to deal with concurrent and hard-real-time software.

The fifth assumption is that the virtual machine provided by the language processor, hardware and operating system will not differ from its specified operation in a manner that affects the analysis. In practice, this assumption means that either the program under analysis avoids any problems with the virtual machine, or that the developers have corrected these problems prior to application of the region analysis technique on the application software. It is *not* assumed that the virtual machine is an ideal machine, nor that it has limitless precision and storage, so that the technique may analyze faults that lead to problems revealed by virtual machine limits.

The last assumption is that undefined variables will not be coincidentally correct. We introduce this assumption to avoid system-specific probabilistic arguments as to the contents of memory. In practice, this assumption introduces a degree of overestimation into the failure region analysis, but in this first formulation of the failure region analysis technique, we view that overestimation as tolerable.

3.2 The Fault and Program Models

In this paper, a fault is an erroneous, possibly non-contiguous, portion of the program source code. A fault may be undesirable extra statements, a miswritten set of statements, or a lack of needed statements. A fault is characterized for the region-analysis technique by three pieces of information. The first is its location, which is either the statement most closely enclosing the fault or the first statement encountered in the course of execution that uses values affected by the fault. Of these two alternatives, the first is most tractable in analysis, but can't be defined for some missing-code faults. Should there be no statement that uses values affected by the erroneous code portion, then if that section causes the lack of an output, the analysis treats it as a self-revealing fault (see section 3.3); otherwise if that section does not cause the lack of an output, then it is not a fault to be analyzed by this technique.

The second piece of information used to characterize a fault is the list of variables that the immediate execution of the fault gives erroneous values to, without any intervening action by any other section of the source code. These erroneous values (henceforth called *errors*), are viewed as being stored in atomic variables, slices of arrays or files with known limits, individual fields of a record structure, indirect-referenced data with all aliases known, and entire sets, strings or variant structures. Should a fault not produce an error, but otherwise cause the program to fail, the analysis treats it as a self-revealing fault.

The third piece of information used to characterize a fault is the conditions under which it will generate an error. These conditions are represented as a boolean expression on the local execution state of the module containing the fault's location. The analysis should include the known limits on slices of arrays or files in this expression.

More formally, then, the analysis technique uses a characterization of a fault as a 3-tuple:

$$< L_F, V_F, C_F >$$

where L_F is the location of the fault, V_F is the list of variables that form the error caused by the fault and C_F is the the conditions under which the fault causes the error, described by a Boolean expression. In practice, this characterization means that the fault must be identified specifically prior to the analysis, but not necessarily corrected. This characterization is not unique, as equivalent Boolean expressions to any C_F may be constructed by negation (i.e., $\neg(\neg C_F) \equiv C_F$). Such an equivalence will not change the results of the analysis technique.

A program is characterized by the variables, modules and statements that are its constituent parts. Modules and variables are identified by unique names (if necessary, by associating them with their defining scope). Statements are identified by their order of occurrence within the program source code, and may enclose subsidiary statements. The input and output statements of the program are explicitly identified. Formally, then, a program is characterized as a 6-tuple:

$$< V_P, R_P, M_P, S_P, I_P, O_P >$$

where V_P is the list of variables defined by the program, R_P is a subset of V_P indicating variables that are input, M_P the list of modules, each having the same structure as the program, S_P is the list of statements, I_P the subset of S_P that are input statements, O_P the subset of S_P that are output statements. An input statement is defined as one where variables receive values from a source external to the program. An output statement is defined as one where messages and variable values are transmitted to a sink external to the program. The initial statement of a program is not considered an input statement, nor is the final statement of the program considered an output statement, unless these statements are members of I_P or O_P, respectively.

A statement S is formally characterized as a 7-tuple:

$$< T_S, D_S, R_S, U_S, C_S, E_S, E'_S >$$

T_S is the text of the statement, exclusive of any enclosed statements. D_S is a list of the variables that are defined (given values) by the statement. R_S is a list of the variables that are referenced by the statement. U_S is a list of the variables that become undefined (lose their value) at statement S. C_S is the portion of T_S that forms a boolean condition. E_S and E'_S are lists of statements enclosed by the statement and executed if C_S evaluates to true or false, respectively. Any or all of D_S, R_S, E_S and E'_S may be the empty list. Should the statement not contain a boolean condition, C_S will be 'true'. A path is a set of statements, a subset of the union of all statements in the program and its submodules. The elements of a path are all statements visited during some specific portion of a program execution, and some elements of a path may also appear as substatements of other elements of the path.

Lastly, the type `BooleanExpression`, used in the sections following, is an expression in the first-order predicate calculus, maintained as an expression rather than being reduced to a Boolean value.

3.3 Overview of the Analysis Technique

This paper describes the failure regions analysis technique using a pseudocode similar to Ada. This description is for compactness and formality, rather than for implication of automatic implementation of the analysis technique.

In a broad view, the analysis technique is made up of four steps: 1) deriving reachability conditions, 2) merging error-generation conditions obtained during fault identification, 3) deriving propagation conditions and 4) post-processing to simplify the region conditions and eliminate non-input variables. A process-program arrangement of these steps appears in Figure 2. A self-revealing fault, one that always causes an immediately visible result that contradicts the specification (e.g., an abnormal program exit) or that always prevents a specified result, omits the analysis of error propagation conditions (i.e., its propagation condition is 'true').

```
procedure DeriveRegion (in $F$:Fault; in $P$:Program;
        out region:BooleanExpression) is
r: BooleanExpression := 'true'
q: BooleanExpression := 'true'
  for each $i \in I_P$ loop
    for each path $t$ from $i$ to $L_F$ loop
      r := r $\vee$ Reach($t, L_F$)
    end loop
  end loop
  r := r $\wedge C_F$  -- obtained from debugging
  if F  is not a self-revealing fault then
    for each $o \in O_P$ loop
      for each path $t$ from $L_F$ to $o$ loop
        q := q $\vee$ Propagate($F, t, o$)
      end loop
    end loop
  end if
  r := Reduce(r $\wedge$ q, $V_P - R_P$)
  region := Simplify(r)
end DeriveRegion
```

Figure 2: Top-Level Description of Region Analysis

The technique requires the user to select paths for the derivation of the reachability and propagation conditions. The assumptions of soft-real-time program character and isolation of faults combine to ensure that the analysis will terminate (i.e., that it need not consider infinite or excessive looping prior to the fault location).

3.4 Analysis for Fault Reachability Conditions

The first analysis step derives the fault's reachability conditions along a specific path. The analysis invokes this step iteratively, as described in figure 2. This step forms a composition of all the conditions connecting one statement in the path to another. One group of conditions occur in the text of `loop`, `if`, and multiple-branch conditional (`case`) statements. This group will henceforth be called *referencing clauses.*

Another group of conditions, henceforth called *defining clauses*, occur due to the effect of assignment statements and procedure or function calls. These clauses represent the changes in variable values that occur during program execution. The analysis uses defining clauses to record transformation of the program state, so that the analyst may change conditions that occur in the program (involving a mix of input values, constants and current values of local variables) to conditions that involve only input values and constants. It is the latter conditions that form the desired bounds on the failure region. Most defining clauses will not appear in the final failure region bounds, but only in the intermediate forms leading to those final bounds.

The reachability condition analysis, then, traverses the statements in the path, forming a composition of the defining and referencing clauses. This traversal is described in figure 3. The formation of the composition is guided by the known location of the fault, and simplified to deal with that specific set of code. Loops in the path are dealt with by applying the exit condition (if the loop body is not in the path) or applying the loop effects (which will be represented by `Effect`(s, c) below, where s is a loop statement operating in a state characterized by c) and then extracting the loop body from the path. If the fault location is part of the loop body, then the traversal algorithm applies the loop effects to emulate any iterations prior to the fault and extracts any paths through the loop body except from the loop initiation to the fault location. Other control statements are dealt with by incorporating their enclosed conditions in the reachability condition and traversing their enclosed statements (if any). Non-control statements are dealt with using the Semantics operation.

```
function Reach (in $t$:path; in $g$:statement)
          returns BooleanExpression is
cur: BooleanExpression := 'true'
flat: set of statements := $\emptyset$
flatn: set of statements := $\emptyset$
  for each statement $s \in t$ loop
    flat := flatten($E_s$)
    flatn := flatten($E'_s$)
    if $s$ is a loop $\wedge$ $E_s \subset t \wedge g \in$ flat then
      cur := cur $\wedge$ Effect($s$,cur) $\wedge C_s$
      $t$ := $(t-$flat$) \cup \{s' \mid s' \in$ flat$\wedge$ $g \in$ flatten$(s')\}$
    elseif $s$ is a loop $\wedge$ $g \in$ flat then
      cur := cur $\wedge$ $C_s$
    elseif $s$ is a loop $\wedge$ $E_s \subset t$ then
      cur := cur $\wedge$ Effect($s$,cur)
      $t$ := $t - E_s$
    elseif $s$ is a loop $\wedge$ $E_s \not\subset t$ then
      cur := cur $\wedge$ $\neg C_s$
    elseif $E_s \neq \emptyset \wedge (E_s \subset t \vee g \in$ flat) then
      cur := cur $\wedge$ $C_s$
    elseif $E'_s \neq \emptyset \wedge (E'_s \subset t \vee g \in$ flatn) then
      cur := cur $\wedge$ $\neg C_s$
    else cur := Reduce(cur, $D_s$) $\wedge$ Semantics($T_s$,cur)
    end if
  end loop
  return Simplify(cur)
end Reach
```

Figure 3: Description of Reachability Analysis

The derivation of the reachability and propagation conditions make use of an operation referred to as Semantics and an inclusion-expanding function flatten. Semantics represents the operation of the underlying virtual machine. The analyst is assumed to take the text of a statement and a context for evaluation to obtain a Boolean expression describing the results of evaluation. Methods for implementing Semantics in practice include symbolic execution[8] and code reading[9].

The function flatten takes a list of statements and returns the union of that list with all of the statements enclosed by any element of that list. The algorithm for flatten is given in figure 4.

```
function flatten(in $L$: set of statements)
        returns set of statements is
cur: set of statements := $\emptyset$
working: set of statements := $L$
  while working $\neq \emptyset$ loop
    $s \in$ working
    working := (working$-\{s\}) \cup E_s \cup E'_s$
    cur := cur$\cup E_s \cup E'_s \cup \{s\}$
  end loop
  return cur
end flatten
```

Figure 4: Flatten Algorithm

3.5 Analysis for Error Propagation Conditions

After the analysis of reachability conditions and the merger of the error generation conditions that were obtained during fault identification, the next step in the failure region derivation is to determine when (if ever) the value of each output is affected by the fault. Figure 5 provides an overview of the error-propagation analysis.

The starting point for this analysis is a list of variables with erroneous values that are immediate results of the fault (denoted by V_F in figure 5). This list forms the initial 'contamination list' for the analysis. The analysis propagates the contamination list separately along each possible path from the location of the fault to each program output. In each specific path, the analysis expands the contamination list as variables receive values derived from one or more variables in the contamination list. the analysis removes variables from the contamination list as they become undefined or as program statements give them new values that do not derive from variables in the contamination list. If the last output statement in the path references variables in the contamination list, then the analysis returns the path condition as a partial failure region condition. If the output statement does not reference any of the variables in the contamination list, or if the contamination list becomes empty, then the path is not incorporated in the failure region conditions.

3.6 Post-Processing of Analysis

To be a practical source of information for testing, the bounding conditions of a software failure region must be simplified as much as feasible without loss of information and the conditions must involve only variables that are of interest to the testers, typically the program input variables. There are two operations that are used to convert the conjunction of conditions derived from the program text into a useful failure region bound. The `Simplify` operation is a conventional Boolean simplification. The `Reduce` operation takes a Boolean condition (involving both defining and referencing clauses) and a list of variables. It returns a Boolean condition in which each reference to any of the variables on the list has been replaced with the associated definition, and any definition of the variables on the list has been eliminated.

The preceding description of the `Simplify` and `Reduce` operations does not imply that these operations are to be fully automated. In practice, the analyst may need to intervene to replace complex or intertwined clauses with simpler forms derived from the program specification, design or source code. This has been found to be particularly true when the input data control the values of program variables only in an indirect fashion.

3.7 Example of Deriving A Software Failure Region

As an example of deriving failure regions from the source code, consider the Pascal function `AcuteAngle` in figure 6. This function is specified to determine if the angle subtended by a rectangle centered at an arbitrary point (X_T, Y_T) when viewed from the origin (i.e., the largest angle with the vertex at the origin and two of the corners of the rectangle as the endpoints) is less than $\frac{\pi}{2}$ radians. The width (w) and length (l) of the rectangle are parallel to the x and y axes, respectively. (See figure 7 for an illustration of a rectangle and origin layout, where α is the angle to be measured.) This function is derived from

```
function Propagate (in F: Fault; in $t$:path;
        in $o$:statement) returns BooleanExpression is
cur: BooleanExpression := 'true'
contamlist: list of variables := $V_F$
flat: set of statements := $\emptyset$
flatn: set of statements := $\emptyset$
  for each statement $s \in t$ loop
    if contamlist is null then
      cur := false
      exit loop
    end if
    flat := flatten($E_s$)
    flatn := flatten($E'_s$)
    if $s$ is a loop $\wedge E_s \subset t \wedge L_F \in$ flat then
      cur := cur$\wedge$Effect($s$,cur) $\wedge C_s$
      $t$ := ($t-$flat) $\cup \{s' \mid s' \in$ flat$\wedge L_F \in$ flatten($s'$)$\}$
    elseif $s$ is a loop $\wedge L_F \in$ flat then
      cur := cur$\wedge C_s$
    elseif $s$ is a loop $\wedge E_s \subset t$ then
      cur := cur$\wedge$ Effect($s$,cur)
      $t$ := $t - E_s$
    elseif $s$ is a loop $\wedge E_s \not\subset t$ then
      cur := cur$\wedge\neg C_s$
    elseif $E_s \neq \emptyset \wedge (E_s \subset t \vee L_F \in$ flat) then
      cur := cur$\wedge C_s$
    elseif $E'_s \neq \emptyset \wedge (E'_s \subset t \vee L_F \in$ flat) then
      cur := cur$\wedge\neg C_s$
    else
      cur := Reduce(cur, $D_s$) $\wedge$ Semantics($T_s$,cur)
    end if
    if contamlist$\cap R_s \neq \emptyset$ then
      contamlist := contamlist$\cup D_s$
    elseif contamlist$\cap D_s \neq \emptyset$ then
      contamlist := contamlist - $D_s$
    end if
    contamlist := contamlist - $U_s$
  end loop
  if contamlist$\cap R_o = \emptyset$ then return false
  else return Simplify(cur)
  end if
end Propagate
```

Figure 5: Description of Propagation Analysis

code developed for a software experiment[15]. The particular implementation in figure 6 has a fault at line 13, where the corners used to calculate the subtended angle are those identified as B and C in figure 8, rather than the proper B and D. Assume that the only input statement is at line 1 and the only output statement is at line 29. The conditions for reaching line 13 are:

$$\begin{aligned}
&X_A = XT - \tfrac{w}{2} \wedge Y_A = YT + \tfrac{l}{2} \wedge \\
&X_B = XT + \tfrac{w}{2} \wedge Y_B = YT + \tfrac{l}{2} \wedge \\
&X_C = XT - \tfrac{w}{2} \wedge Y_C = YT - \tfrac{l}{2} \wedge \\
&X_D = XT + \tfrac{w}{2} \wedge Y_D = YT - \tfrac{l}{2} \wedge \\
&Y_D < 0 \wedge Y_B > 0 \wedge X_D < 0
\end{aligned}$$

The first four lines are defining clauses, the last line a referencing clause.

The conditions under which the code produces an erroneous value due to the substitution of C for D are those where the angle formed by B, C, and the origin (angle C0B) is different from B, D and the origin (angle D0B), or:

$$\frac{Y_C}{X_C} > \frac{Y_D}{X_D}$$

which, since $Y_C = Y_D$, simplifies to $X_D > X_C$, which reduces to $w > 0$. Lastly, the conditions under which the erroneous value isn't masked by the comparison at line 29 in figure 6 are where angle C0B is less than $\frac{\pi}{2}$ and angle D0B is greater than $\frac{\pi}{2}$, or:

$$\left(\tan^{-1}\left(\frac{Y_B}{X_B}\right) - \tan^{-1}\left(\frac{Y_C}{X_C}\right)\right) \leq \frac{\pi}{2} < \left(\tan^{-1}\left(\frac{Y_B}{X_B}\right) - \tan^{-1}\left(\frac{Y_D}{X_D}\right)\right)$$

Combining these inequalities, substituting for defining clauses and simplifying yields the result:

$$\begin{gathered}
\left(\tfrac{-l}{2} < YT < \tfrac{l}{2}\right) \wedge X_T > \tfrac{-w}{2} \wedge w > 0 \wedge \\
\left(\tan^{-1}\left(\frac{YT+\frac{l}{2}}{XT+\frac{w}{2}}\right) - \tan^{-1}\left(\frac{YT-\frac{l}{2}}{XT-\frac{w}{2}}\right)\right) \leq \frac{\pi}{2} < \left(\tan^{-1}\left(\frac{YT+\frac{l}{2}}{XT+\frac{w}{2}}\right) - \tan^{-1}\left(\frac{YT-\frac{l}{2}}{XT+\frac{w}{2}}\right)\right)
\end{gathered}$$

A cross-section of this region appears as the cross-hatched portion of figure 8 for a particular set of values for X_T, Y_T, w and l. As we vary those four values, the failure region forms a tetrahedral cylinder through the four-dimensional input space, with a rotating tetrahedral groove through one face.

```
function AcuteAngle(XT,YT,W,L:real):boolean;
const Pi=3.1415926535;
var AX,AY, BX,BY, CX,CY, DX,DY: real;
   Angle, HalfWidth, HalfLength: real;
begin
   HalfWidth := W / 2; HalfLength := L / 2;
   AX := XT-HalfWidth; AY:=YT+HalfLength;
   DX := XT+HalfWidth; DY:=YT-HalfLength;
   BX := DX;           BY:=AY;
   CX := AX;           CY:=DY;
   if ((YO > BY) and (XO > BX)) then
      Angle := abs(arctan(DY/DX) - arctan(AY/AX))
   else if ((XO < CX) and (YO < CY)) then
      Angle := abs(arctan(AY/AX) - arctan(DY/DX))
   else if ((YO > AY) and (XO < AX)) then
      Angle := abs(arctan(CY/CX) - arctan(BY/BX))
   else if ((YO > DY) and (XO > DX)) then
      Angle := abs(arctan(BY/BX) - arctan(CY/CX))
   else if XO > BX then
      Angle := abs(arctan(DY/DX) + arctan(BY/BX))
   else if XO < AX then
      Angle := abs(arctan(AY/AX) + arctan(CY/CX))
   else if (YO >= AY) then
      if (XO = AX) then Angle := Pi/2 - abs(arctan(BY/BX))
      else if (XO < BX) then
         Angle := Pi - abs(arctan(AY/AX)) - abs(arctan(BY/BX))
      else Angle := Pi/2 - abs(arctan(AY/AX))
   else if (YO <= CY) then
      if (XO = DX) then Angle := Pi/2 - abs(arctan(CY/CX))
      else if (XO > CX) then
         Angle := Pi - abs(arctan(DY/DX)) - abs(arctan(CY/CX))
      else Angle := Pi/2 - abs(arctan(DY/DX))
   else  Angle := 2 * Pi;
   AcuteAngle := Angle <= (Pi/2);
end;
```

Figure 6: Function AcuteAngle

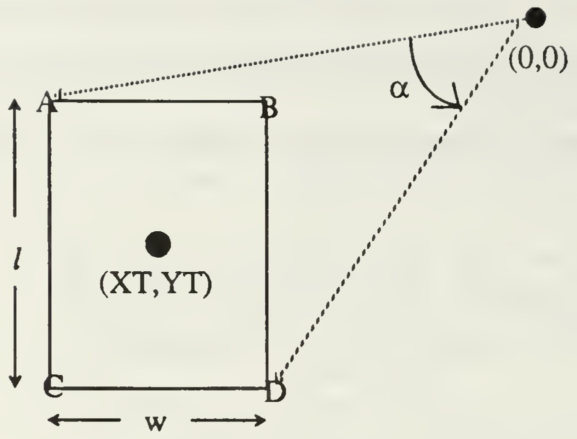


Figure 7: Parameter Layout for Function AcuteAngle

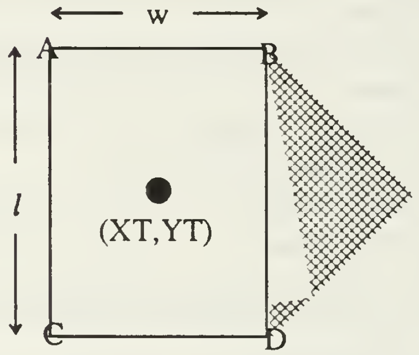


Figure 8: Failure Region in Function AcuteAngle

3.8 Validation of the Analysis Technique

Section 1 defined a failure region as bounded by the necessary and sufficient conditions for a fault being executed and leading to a failure. This section proves the following theorem:

Theorem 1 *Given the assumptions in section 3.1 and that the fault detection process accurately supplies L_F, V_F and C_F, and that the* `Semantics` *function accurately supplies the effect of its argument statement on its argument environment,* `DeriveRegion`*, if it terminates, will produce necessary and sufficient conditions for executing a fault and causing it to lead to any failure.*

This theorem is the conjunction of the following two propositions:

Proposition 1 (Necessity) *All input sequences that satisfy the conditions produced by* `DeriveRegion` *will cause the fault to be executed and lead to some failure.*

Proposition 2 (Sufficiency) *There are no input sequences that fail to satisfy the conditions produced by* `DeriveRegion` *and cause the fault to be executed and lead to any failure.*

Both of these propositions may be conveniently proved by contradiction. The negation of Proposition 1 assumes that some input satisfies the conditions generated by `DeriveRe-gion` but does not execute the fault and lead to a failure. This may only happen due to one or more of the following:

n1 The input does not cause the code at the fault location to be executed.

n2 The input does not cause the fault to generate an error.

n3 The input does not execute a path from the fault to any output statement.

n4 The input maps the error generated by the fault into a correct state along the path to each output statement.

Option n1 may occur iff at some statement prior to L_F, the condition on the branch that leads to L_F was false in the state derived from the input. That implies that there exists some input statement $i \in I_P$ such that a path from i to L_F exists, but that path is not included in the derivation of failure region bounds, or the conditions along the path were incorrectly analyzed by `Reach`. The implication that the path was not included is in conflict with the 'all paths from i to L_F' loop in `DeriveRegion`.

The implication that the conditions along a path were incorrectly analyzed by `Reach` implies two possibilities: the referencing clauses were incorrectly analyzed or the defining clauses were incorrectly analyzed. The possibility that the defining clauses were incorrectly analyzed is a contradiction of the assumption that `Semantics` accurately captures the effects of the statement. The possibility that the referencing clauses were incorrectly analyzed is shown a contradiction by examination of the condition handling in `Reach` (the **`if`** structure inside of the **`for each`** loop).

In any path through the text of a program, there are three possible traversals of a single-entry, single-exit section of the text : the section could be completely traversed, the path could terminate inside of the section, or the path could include none of the statements in the section. Complete traversal of a code section is dealt with by the third, fifth and sixth **`if`** clauses in `Reach`, in which the conditions for entry of the code section are included in the path conditions. Partial traversal of a code section is dealt with by the first, second, fifth and sixth **`if`** clauses, where the presence of the path termination is tested, and if present, the conditions for entry of the code section are included in the path conditions. Non-traversal of a code section is dealt with by the **`for each`** loop, which restricts construction of path conditions to the statements in the path, and the fourth **`if`** clause, which incorporates the loop exit conditions if the body is not executed. Thus in each case the appropriate conditions are included in the path condition, so incorrect analysis by `Reach` is shown to be a contradiction.

Option n2 is a contradiction of the assumption that C_F has been accurately captured by the fault detection process, and hence may be neglected.

Option n3 may occur iff at all statements reachable from L_F, the branches that include any member of the set of program output statements required conditions that were found to be false in the state derived from the input. That implies that there exist paths from L_F to at least one output statement o, but these paths are not included in the derivation of the failure region bounds or the conditions along the path were incorrectly analyzed by `Propagate`. The implication that the path was not included is in conflict with the 'all paths from L_F to o' loop in `DeriveRegion`. The implication that the conditions along the path were incorrectly analyzed by `Propagate` may be shown to be a contradiction through argument paralleling the preceding discussion of `Reach`.

Option n4 may occur iff at some point on the path selected by the input after the execution of L_F, all variables affected by the fault and subsequent erroneous calculation have had their values replaced by values resulting from calculation or reference with solely unaffected variables. This conflicts with the first **`if`** inside the **`for each`** loop in `Prop-agate`, which, by returning 'false', precludes input that selects such paths from inclusion in the failure region.

Since each possible cause of the negation of Proposition 1 has been shown to be a contradiction, that proposition is found to be true. `DeriveRegion` is thus proven to generate necessary conditions for a fault to be executed and lead to a failure.

The negation of Proposition 2 assumes that some input exists that does not satisfy the conditions generated by `DeriveRegion` but does lead the fault to produce some failure. This may only happen through one or more of the following:

s1 The input causes program execution along a path from an input statement through the fault location that was not considered in `DeriveRegion`.

s2 The conditions derived from the input cause the fault to generate an error in a manner not considered in C_F.

s3 The input causes program execution along a path from the fault location to an output statement that was not considered in `DeriveRegion`.

Option s1 is shown to be a contradiction through an argument that parallels the refutation of option n1. Option s2 is a direct contradiction of the assumption that C_F has been accurately captured by the fault detection process. Option s3 is shown to be a contradiction through an argument that parallels the refutation of option n3. All three options being shown as contradictions, `DeriveRegion` is proven to generate sufficient conditions for a fault to be executed and lead to a failure.

This proof of necessary and sufficient conditions is subject to the assumptions made at the start of the section. These assumptions limit the applicability of the manual technique to those contexts that can reasonably satisfy them. If the technique is applied in a context where these assumptions do not hold, it may generate the necessary and sufficient conditions, but this proof does not hold.

4 Automated Support for Failure Region Analysis

While the manual process allows for the derivation of software failure regions, it is too mentally taxing and too slow for use with large or complex software. To facilitate and accelerate failure region derivation, five semiautomated software tools have been developed to support the failure region analysis: `REACHER`, `WALKER`, `FALTER`, `SPACER` and `VIEWER`. A flow diagram of the use of these tools is in figure 9.

These tools are not intended to replace a human analyst, but rather to support the analyst in the derivation of failure regions. At any point in the analysis process, the analyst may intervene to replace conditions, change the source code fragment being analyzed, or

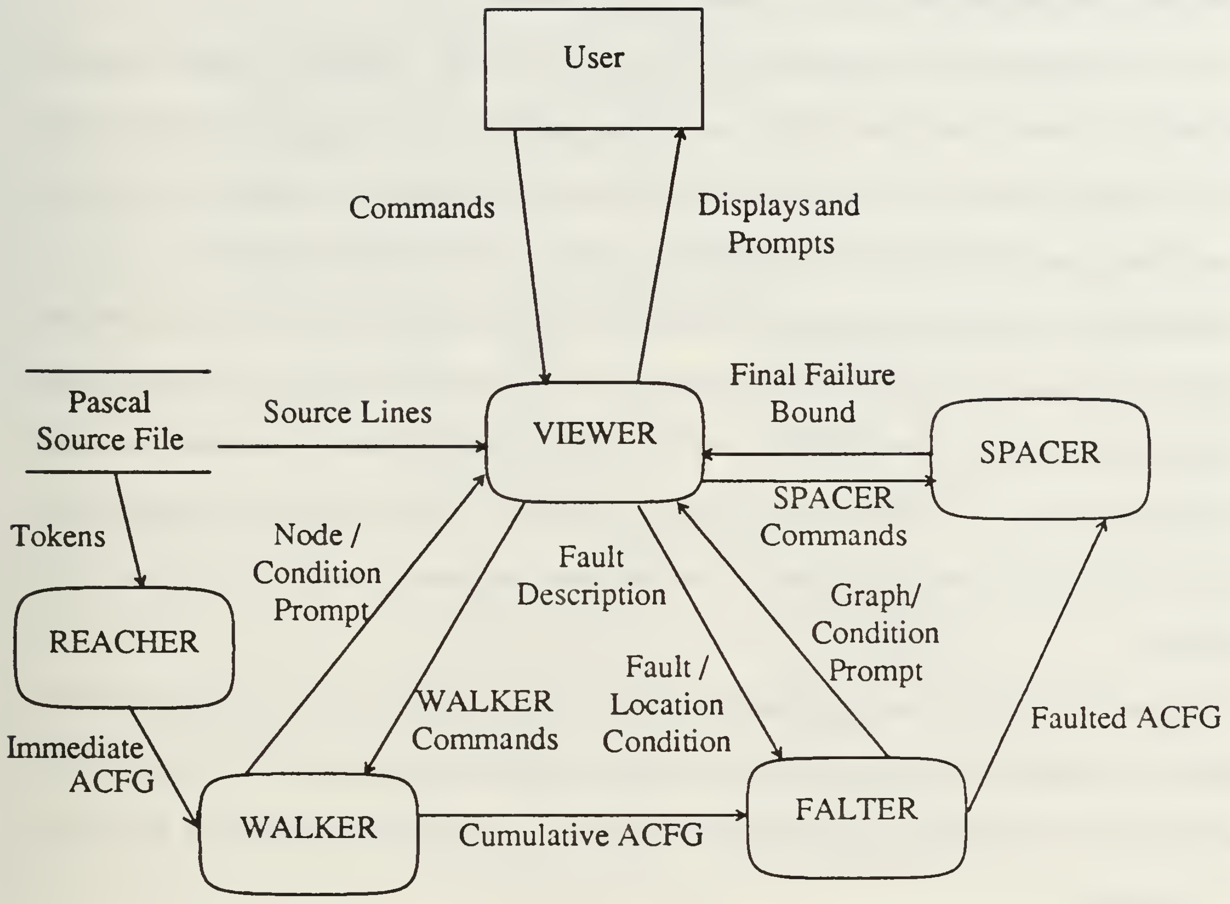


Figure 9: Flow Diagram of Support Tools

examine previous analysis results. At some points in the process, the analyst is required to provide information that is not automatically determinable.

Two tools support the reachability analysis process. REACHER parses Pascal programs and builds a control flow graph. This control-flow graph is a data-structure representation of the program model given in section 3.2. Using the control-flow graph, the conditions under which each statement in the program may be reached may be derived.

REACHER is constructed using LEX and YACC and a specification for Pascal based on the ISO standard. In a single non-interactive pass over the Pascal source code, REACHER creates a list of blocks corresponding to each procedure and function. Each block contains information such as the declaration text for each block, the sub-blocks belonging to or nested within each block and a pointer tracing the block back through the thread of its definition.

Concurrent with the construction of the linked list of blocks is the construction of a linked list of nodes that correspond to the statements in the program. It is this list that results in the control-flow graph. For example, consider the Pascal fragment in Figure 10.

```
begin
   readln(x);
   if x > 3 then
      y := 10
   else
      y := 20;
   writeln(x, y)
end
```

Figure 10: Example Pascal Fragment

The list would contain nodes for **begin-end**, the `readln` procedure call, the **if-then-else**, the two assignment statements and the `writeln` procedure call. The nodes are linked as shown in figure 11, forming the control-flow graph. Each node is annotated with the conditions under which program flow may proceed to each succeeding statement.

Once REACHER has constructed the initial control-flow graph, annotated with the im-

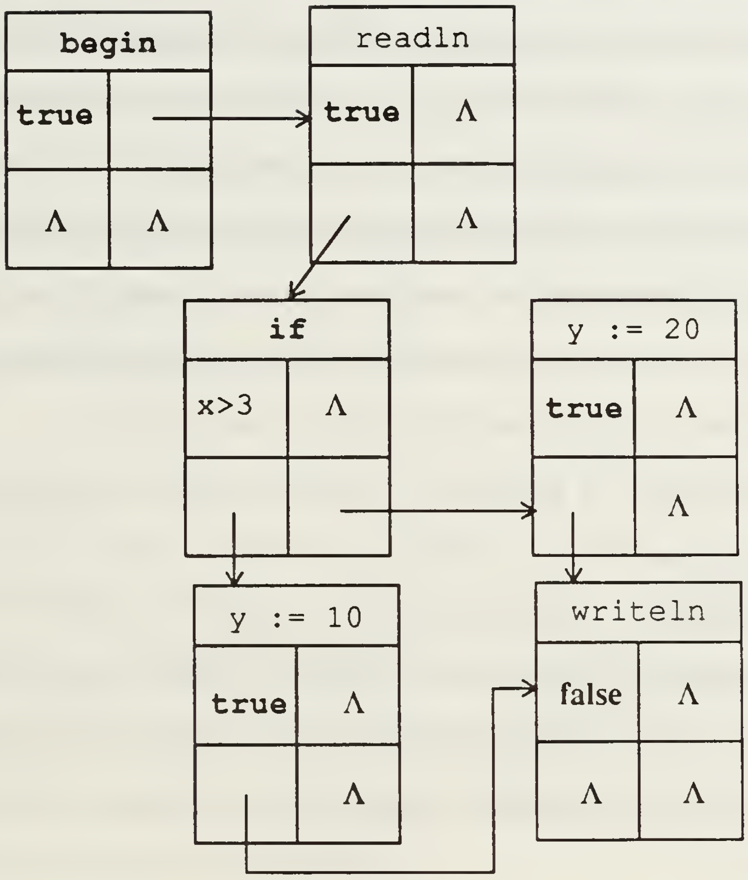


Figure 11: Node Linkages in Example Pascal Fragment

mediate branching conditions, WALKER is used to collapse unneeded nodes and to collect the immediate branching conditions into the node reachability conditions. Unneeded nodes are those unrelated to the fault being analyzed. They are collapsed by merger of the node information in the internal data structure. The collection of branching conditions into reachability conditions is accomplished by a user-guided loop [0,1] traversal as described in section 3.4. By default WALKER will 'and' together the branching conditions of the nodes traversed. If the user finds the resultant conditions undesirable, WALKER allows the user to replace the current condition with one of the user's choosing at any point in the traversal. WALKER is implemented in C using the BSD Unix support libraries.

FALTER is used to attach a description of a known fault to the data structure corresponding to the program block in which the fault is found. The fault description corresponds to the information described in the fault model in section 3.2, including the error-generation conditions and the list of variables with values immediately contaminated by the fault. FALTER then translates the internal structure into a LISP expression for symbolic execution by SPACER. FALTER is implemented in C using the BSD Unix support libraries.

SPACER supports the propagation analysis described in section 3.5. At the start of execution, SPACER prompts the analyst to select the fault to be analyzed from the list of faults identified through FALTER in the current program. SPACER then symbolically executes the program, performing a traversal of the program paths. When the fault location is reached, SPACER initiates the analysis process described in section 3.5, using the information provided via FALTER. At each output statement, SPACER reports the current propagation conditions for the variables referenced in the output. During this process, SPACER requires user guidance to select paths to traverse and to resolve complex expressions. SPACER periodically reports to the user the current state of the analysis and the statements being traversed. SPACER is implemented in Allegro Common LISP, using portions of the UNISEX symbolic execution package[7].

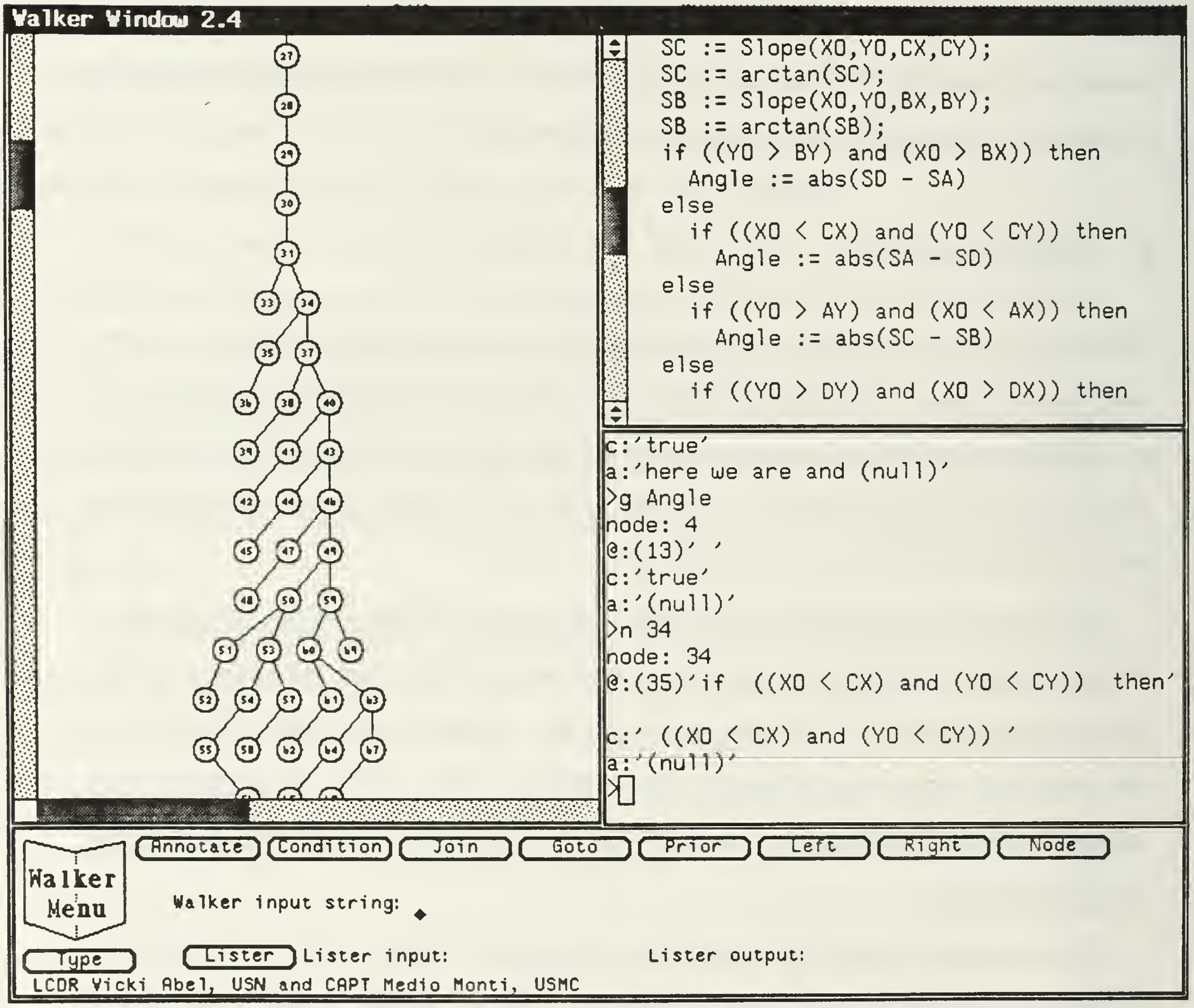
While each of the above tools may be directly accessed by the analyst, a coordinating visual interface is provided by the VIEWER tool[2]. VIEWER presents textual and graphic information to facilitate the analyst's tracking of the analysis process, provides a mouse-driven command structure to replace the text command structure supported by the other tools, deals with tool initiation and termination, eases consistency between tool usage, and provides an extensible command structure to ease common commands. VIEWER is implemented in C using the SunView user interface libraries. A sample of VIEWER operation (one supporting WALKER execution) is given in figure 12.

5 Conclusions

This paper has presented a new technique for deriving the bounding conditions for software failure regions. These conditions form the necessary and sufficient conditions for an individual program fault to be executed and to lead to a failure. But the technique requires that the fault be known prior to the analysis. One testing task in which this knowledge would be present is the regression testing task.

There are two immediately obvious ways in which failure regions analysis aids regression testing. First, the failure region offers a framework within which statistical sampling techniques can be used to evaluate fault correction. Testers, if they know the limits on the useful test cases, can intelligently plan regression tests, rather than iterate previous testing. Research is in progress on detailed regression-testing techniques based on sampling of failure regions.

The second way failure regions analysis aids regression testing is by focusing detailed attention on the fault. As a review method, failure regions analysis may allow the analyst to identify if a proposed correction is inadequate prior to run-time testing. As a planning method, failure regions analysis may allow maintenance personnel further insight on the fault as they consider alternative corrections to the fault.



```
SC := Slope(X0,Y0,CX,CY);
SC := arctan(SC);
SB := Slope(X0,Y0,BX,BY);
SB := arctan(SB);
if ((Y0 > BY) and (X0 > BX)) then
  Angle := abs(SD - SA)
else
  if ((X0 < CX) and (Y0 < CY)) then
    Angle := abs(SA - SD)
else
  if ((Y0 > AY) and (X0 < AX)) then
    Angle := abs(SC - SB)
else
  if ((Y0 > DY) and (X0 > DX)) then
```

```
c:'true'
a:'here we are and (null)'
>g Angle
node: 4
@:(13)' '
c:'true'
a:'(null)'
>n 34
node: 34
@:(35)'if  ((X0 < CX) and (Y0 < CY))  then'

c:' ((X0 < CX) and (Y0 < CY)) '
a:'(null)'
>
```

Figure 12: Sample of VIEWER Operation

Finally, knowledge of the failure region bounds allows examination of a variety of issues relating to the applicability of software testing techniques. Research is in progress that applies software failure region analysis to a population of software with known faults and examines these issues. In summary, software failure regions analysis offers both assistance in portions of the software test effort and insight into a variety of phenomena relating to faults and their detection.

6 Acknowledgements

The authors wish to thank Richard Kemmerer for his provision and permission to use UNISEX as a portion of the `SPACER` tool. MAJ Medio Monte, USMC and LCDR Vicki Sue Abel, USN, put in long hours of quality work to construct the `VIEWER` tool. Many helpful comments during the conception and writing of this paper were provided by Dr. Richard Hamming, Dr. Man-tak Shing and LT Lelon Ginn, USN.

References

[1] *Glossary of Software Engineering Terminology*, ANSI-IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, 1983.

[2] Abel, V.S. and Monte, M., *VIEWER: A User Interface for Failure Region Analysis*, Master's Thesis, Computer Science Dept., Naval Postgraduate School, Monterey CA, December 1990.

[3] Ammann, P.E. and Knight, J.C., "Data Diversity: An Approach to Software Fault Tolerance", *IEEE Transactions on Computers*, April 1988, pp. 418–425.

[4] Dunham, J.R., and Finelli, G.B., "Real-Time Software Failure Characterization", *Proceedings of the Fifth Annual Conference on Computer Assurance*, Gaithersburg, MD, June 1990, pp. 39–45.

[5] Gayen, J.-T. and Kuchta, D., "Possibilities and Limitations of Error Detection by White-box Testing Methods, Including the Domain Borders Method", *Proceedings of the Safety of Computer Control Systems Symposium 1988 (SAFECOMP '88)*, Fulda, FRG, November 1988, pp. 35–40.

[6] Ginn, L.L., *An Empirical Approach to Analysis of Similarities Between Software Failure Regions*, Master's Thesis, Computer Science Dept., Naval Postgraduate School, Monterey CA, September 1991.

[7] Kemmerer, R. A. and Eckmann, S. T., "UNISEX: A UNIx-based Symbolic EXecutor for Pascal", *Software – Practice and Experience*, Vol 15, No. 5, May 1985, pp. 439–455.

[8] King, J.C., "Symbolic Execution and Program Testing", *Communications of the ACM*, Vol 19, No. 7, July 1976, pp. 385–394.

[9] Linger, R. C., Mills, H. D. and Witt, B.I., *Structured Programming, Theory and Practice*, Addison-Wesley, 1979.

[10] Myers, G. J., *The Art Of Software Testing*, John Wiley and Sons, New York, NY, 1979.

[11] Rapps, S. and Weyuker, E. J., "Selecting Software Test Data using Data Flow Information", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, April 1985, pp. 367–375.

[12] Richardson, D.J. and Thompson, M. C., "The RELAY Model of Error Detection and Its Application", *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, 1988, pp. 223–230.

[13] Shimeall, T.J., "FALTER–A Fault Annotation Tool", Technical Report NPS52-89-0-51, Naval Postgraduate School, Monterey, CA, September, 1989.

[14] Shimeall, T.J., "REACHER–A Reachability Condition Derivation Tool", Technical Report NPS52-89-050, Naval Postgraduate School, Monterey, CA, September, 1989.

[15] Shimeall, T.J. and Leveson, N.G., "An Empirical comparison of Software Fault Tolerance and Fault Elimination", *IEEE Transactions on Software Engineering*, Vol. SE-17, No. 2, February 1991, pp. 173–182.

[16] Voas, J. M. and Morell, L. J., "Fault Sensitivity Analysis (PIA) Applied to Computer Programs", Technical Report WM-89-4. College of William and Mary, Williamsburg, VA, 10 December 1989.

[17] Watson, G. F., "Bell companies, manufacturers join to cure phone outages", *The Institute*, vol 15, No. 7, September 1991, p. 1, 7.

[18] White, L.J. and Wiszniewski, B., "Complexity of Testing Iterated Borders for Structured Programs", *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, 1988, pp. 231–237.

Distribution List

Defense Technical Information Center, Cameron Station, Alexandria, VA 22314	2 copies
Library, Code 0142 Naval Postgraduate School, Monterey, CA 93943	2 copies
Center for Naval Analyses, 4401 Ford Avenue Alexandria, VA 22302-0268	1 copy
Director of Research Administration, Code 012, Naval Postgraduate School, Monterey, CA 93943	1 copy
Dr. Timothy Shimeall Naval Postgraduate School, Code CS/Sm, Dept. of Computer Science Monterey, California 93943-5100	15 copies
Robert E. Westbrook Code 31C Embedded Computing Technology Office Naval Weapons Center China Lake, CA 93555-6001	2 copies
CDR Rachel Griffin Naval ROTC University of Rochester Rochester, NY 14627	1 copy
MAJ J. Manning Bolchoz 1025 Chambers Lane Mount Pleasant, SC 29464	1 copy